24-依赖倒置原则：高层代码和底层代码，到底谁该依赖谁？

你好！我是郑晔。

上一讲，我们讲了ISP原则，知道了在设计接口的时候，我们应该设计小接口，不应该让使用者依赖于用不到的方法。但在结尾的时候，我留下了一个尾巴，说在那个例子里面还有一个根本性的问题：依赖方向搞反了。

依赖这个词，程序员们都好理解，意思就是，我这段代码用到了谁，我就依赖了谁。依赖容易有，但能不能把依赖弄对，就需要动点脑子了。如果依赖关系没有处理好，就会导致一个小改动影响一大片，而把依赖方向搞反，就是最典型的错误。

那什么叫依赖方向搞反呢？这一讲我们就来讨论关于依赖的设计原则：依赖倒置原则。

谁依赖谁

依赖倒置原则（Dependency inversion principle，简称DIP）是这样表述的：

> 高层模块不应依赖于低层模块，二者应依赖于抽象。
> High-level modules should not depend on low-level modules. Both should depend on abstractions.

> 抽象不应依赖于细节，细节应依赖于抽象。
> Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

我们学习这个原则，最重要的是要理解“倒置”，而要理解什么是“倒置”，就要先理解所谓的“正常依赖”是什么样的。

讲结构化编程时，我们曾经说过结构化编程解决问题的思路是自上而下地进行功能分解，这种解决问题的思路很自然地就会延续到很多人的编程习惯中。按照分解的结果，进行组合。所以，我们很自然地就会写出类似下面的这种代码：

```
class CriticalFeature {
  private Step1 step1;
  private Step2 step2;
  ...

  void run() {
    // 执行第一步
    step1.execute();
    // 执行第二步
    step2.execute();
    ...
  }
}
```

但是，这种未经审视的结构天然就有一个问题：**高层模块会依赖于低层模块**。在上面这段代码里，

CriticalFeature类就是高层类，Step1和Step2就是低层模块，而且Step1和Step2通常都是具体类。虽然这是一种自然而然的写法，但是这种写法确实是有问题的。

在实际的项目中，代码经常会直接耦合在具体的实现上。比如，我们用Kafka做消息传递，我们就在代码里直接创建了一个KafkaProducer去发送消息。我们就可能会写出这样的代码：

```
class Handler {
  private KafkaProducer producer;

  void execute() {
    ...
    Message message = ...;
    producer.send(new KafkaRecord<>("topic", message);
    ...
  }
}
```

也许你会问，我就是用了Kafka发消息，创建一个KafkaProducer，这有什么问题吗？其实，这个问题我们在课程中已经讲过了，就是说我们需要站在长期的角度去看，什么东西是变的、什么东西是不变的。Kafka虽然很好，但它并不是系统最核心的部分，我们在未来是可能把它换掉的。

你可能会想，这可是我实现的一个关键组件，我怎么可能会换掉它呢？你还记得吗，软件设计需要关注长期、放眼长期，所有那些不在自己掌控之内的东西，都是有可能被替换的。其实，在我前面讲的很多内容里，你也可以看到，替换一个中间件是经常发生的。所以，依赖于一个可能会变的东西，从设计的角度看，并不是一个好的做法。

那我们应该怎么做呢？这就轮到倒置登场了。

所谓倒置，就是把这种习惯性的做法倒过来，让高层模块不再依赖于低层模块。那要是这样的话，我们的功能又该如何完成呢？计算机行业中一句名言告诉了我们答案：

> 计算机科学中的所有问题都可以通过引入一个间接层得到解决。
> All problems in computer science can be solved by another level of indirection
> —— David Wheeler

是的，引入一个间接层。这个间接层指的就是DIP里所说的抽象。不过，在我们课程里，我一直用的说法是**模型**。也就是说，这段代码里面缺少了一个模型，而这个模型就是这个低层模块在这个过程中所承担的角色。

既然这个模块扮演的就是消息发送者的角色，那我们就可以引入一个消息发送者（MessageSender）的模型：

```
interface MessageSender {
  void send(Message message);
}
```

```
class Handler {
  private MessageSender sender;

  void execute() {
    ...
    Message message = ...;
    sender.send(message);
    ...
  }
}
```

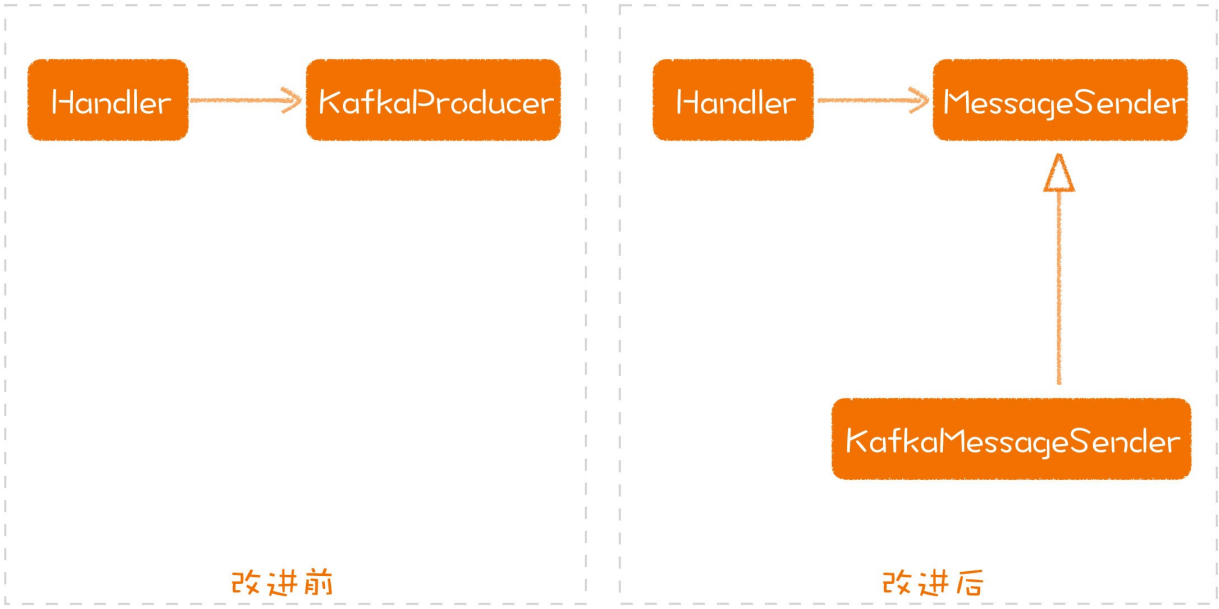
有了消息发送者这个模型，那我们又该如何把Kafka和这个模型结合起来呢？那就要实现一个Kafka的消息发送者：

```
class KafkaMessageSender implements MessageSender {
  private KafkaProducer producer;

  public void send(final Message message) {
    this.producer.send(new KafkaRecord<>("topic", message));
  }
}
```

这样一来，高层模块就不像原来一样**直接**依赖低层模块，而是将依赖关系“倒置”过来，让低层模块去依赖由高层定义好的接口。这样做的好处就在于，将高层模块与低层实现解耦开来。

如果未来我们要替换掉Kafka，只要重写一个MessageSender就好了，其他部分并不需要改变。这样一来，我们就可以让高层模块保持相对稳定，不会随着低层代码的改变而改变。



依赖于抽象

理解了DIP的第一部分后，我们已经知道了要建立起模型（抽象）的概念。

你有没有发现，我们学习的所有原则都是在讲，尽可能把变的部分和不变的部分分开，让不变的部分稳定下来。我们知道，模型是相对稳定的，实现细节则是容易变动的部分。所以，构建出一个稳定的模型层，对任何一个系统而言，都是至关重要的。

那接下来，我们再来分析DIP的第二个部分：抽象不应依赖于细节，细节应依赖于抽象。

其实，这个可以更简单地理解为一点：**依赖于抽象**，从这点出发，我们可以推导出一些更具体的指导编码的规则：

- 任何变量都不应该指向一个具体类；
- 任何类都不应继承自具体类；
- 任何方法都不应该改写父类中已经实现的方法。

我们在讲多态时，提到过一个List声明的例子，其实背后遵循的就是这里的第一条规则：

```
List<String> list = new ArrayList<>();
```

在实际的项目中，这些编码规则有时候也并不是绝对的。如果一个类特别稳定，我们也是可以直接用的，比如字符串类。但是，请注意，这种情况非常少。因为大多数人写的代码稳定度并没有那么高。所以，上面几条编码规则可以成为覆盖大部分情况的规则，出现例外时，我们就需要特别关注一下。

到这里，你已经理解了在DIP的指导下，具体类还是能少用就少用。但还有一个问题，最终，具体类我们还是要用的，毕竟代码要运行起来不能只依赖于接口。那具体类应该在哪用呢？

我们讨论的这些设计原则，核心的关注点都是一个个的业务模型。此外，还有一些代码做的工作是负责把这些模型组装起来，这些负责组装的代码就需要用到一个一个的具体类。

是不是说到这里，感觉话题很熟悉呢？是的，我们在第五讲讨论过DI容器的来龙去脉，在Java世界里，做这些组装工作的就是DI容器。

因为这些组装工作几乎是标准化的，而且非常繁琐。如果你常用的语言中，没有提供DI容器，最好还是把负责组装的代码和业务模型放到不同的代码里。

DI容器在最初的讨论中有另外一个说法叫IoC容器，这个IoC是Inversion of Control的缩写，你会看到IoC和DIP中的I都是inversion，二者表现的意图实际上是一致的。

理解了DIP，再来使用DI容器，你会觉得一切顺理成章，因为依赖之所以可以注入，是因为我们的设计遵循了 DIP。而只知道DI容器不了解DIP，时常会出现让你觉得很为难的模型组装，根本的原因就是设计没有做好。

关于DIP，还有一个形象的说法，称为好莱坞规则：“Don’t call us, we’ll call you”。放在设计里面，这个翻译应该是“别调用我，我会调你的”。显然，这是一个框架才会有的说法，有了一个稳定的抽象，各种具体的实现都应该是由框架去调用。

是的，如果你想去编写一个框架，理解DIP是非常重要的。毫不夸张地说，不理解DIP的程序员，就只能写功能，不能构建出模型，也就很难再上一个台阶。在前面讨论程序库时，我建议每个程序员都去锻炼编写程序库，这其实就是让你去锻炼构建模型的能力。

有了对DIP的讨论，我们再回过头看上一讲留下的疑问，为什么说一开始TransactionRequest是把依赖方向搞反了？因为最初的TransactionRequest是一个具体类，而TransactionHandler是业务类。

我们后来改进的版本里引入一个模型，把TransactionRequest变成了接口，ActualTransactionRequest 实现这个接口，TransactionHandler只依赖于接口，而原来的具体类从这个接口继承而来，相对来说，比原来的版本好一些。

对于任何一个项目而言，了解不同模块的依赖关系是一件很重要的事。你可以去找一些工具去生成项目的依赖关系图，然后，你就可以用DIP作为一个评判标准，去衡量一下你的项目在依赖关系上表现得到底怎么样了。很有可能，你就找到了项目改造的一些着力点。

理解了 DIP，再来看一些关于依赖的讨论，我们也可以看到不同的角度。比如，循环依赖，有人会说从技术上要如何解决它，但实际上，循环依赖就是设计没有做好的结果，把依赖关系弄错了，才可能会出现循环依赖，先把设计做对，把该有的接口提取出来，依赖就不会循环了。

至此，SOLID的五个原则，我们已经讲了一遍。有了前面对于分离关注点和面向对象基础知识的铺垫，相信你理解这些原则的难度也会相应的降低了一些。

你会看到，理解这些原则，关键的第一步还是**分离关注点**，把不同的内容区分开来。然后，用这些原则再把它们组合起来。而当你理解了这些原则，再回头去看，也能加深对面向对象特点的认识，现在你应该更能深刻体会到多态在面向对象世界里发挥的作用了。

总结时刻

今天我们讲了依赖倒置原则，它的表述是：

- 高层模块不应依赖于低层模块，二者应依赖于抽象。
- 抽象不应依赖于细节，细节应依赖于抽象。

理解这个原则的关键在于理解“倒置”，它是相对于传统自上而下的解决问题然后组合的方式而言的。高层模块不依赖于低层模块，可以通过引入一个抽象，或者模型，将二者解耦开来。高层模块依赖于这个模型，而低层模块实现这个模型。

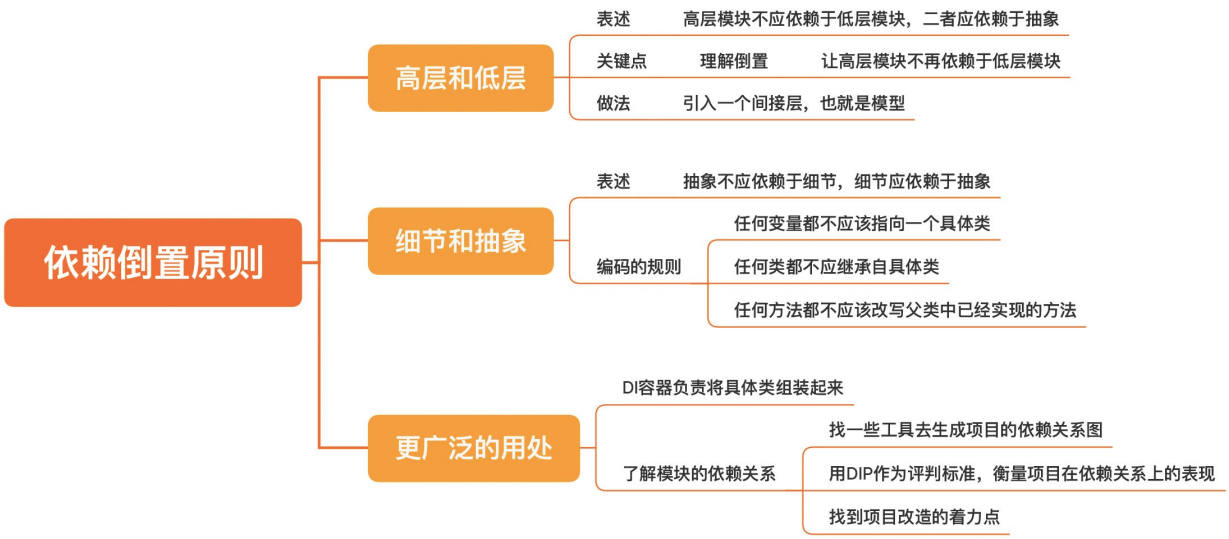
DIP 还可以简单理解成要依赖于抽象，由此，还可以推导出一些指导编码的规则：

- 任何变量都不应该指向一个具体类；
- 任何类都不应继承自具体类；
- 任何方法都不应该改写父类中已经实现的方法。

如果我们的模型都按照DIP去编写，具体类可以放到模型组装的过程去使用，对于 Java 世界而言，这个工作是由 DI 容器完成的。即便是没有 DI 容器的语言，组装代码与模型代码也应该是分开的。把 DIP 应用于项目，可以先从生成依赖关系图开始，找到可以改进的点。

学习了设计原则之后，我们已经有了标准去指导我们的设计，有了尺子去衡量我们的设计。接下来，我们要学习比设计原则更具体的内容：设计模式，下一讲，我们来谈谈如何学习设计模式。

如果今天的内容你只能记住一件事，那请记住：**依赖于构建出来的抽象，而不是具体类**。



极客时间

思考题

最后我想请你去了解一下防腐层（Anti-Corruption Layer），结合今天讲的DIP，谈谈它的适用场景，欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- qinsi 2020-07-22 12:00:35
  “任何类都不应继承自具体类”和“任何方法都不应该改写父类中已经实现的方法”，放在一起是说，类可以继承自抽象类，但不应该改写抽象类中已实现的方法吗？因为如果类都只是依赖和实现接口的话，就没有所谓的继承和父类中的实现了。

- Jxin 2020-07-22 09:58:13
  防腐层可以解耦对外部系统的依赖。包括接口和参数。防腐层还可以贯彻接口隔离的思想，以及做一些功能增强(加缓存,异步并发取值)。

- 阳仔 2020-07-22 09:22:19
  依赖倒置原则说的是：
  1.高层模块不应依赖于低层模块，二者都应依赖于抽象
  2.抽象不应依赖于细节，细节应依赖于抽象
  总结起来就是依赖抽象（模型），具体实现抽象接口，然后把模型代码和组装代码分开，这样的设计就是分离关注点，将不变的与不变有效的区分开

- 大雁小鱼 2020-07-22 09:10:43
  防腐层就是做隔离，一般用于系统集成中，新系统与旧系统集成，为了不让改变扩散，在系统之间多了一个层，叫防腐层，用于控制修改扩散，防腐层也有抽象对方系统的味道。

- 不记年 2020-07-22 08:59:01
  感觉TransactionRequest只是存储了数据, 没必要要在抽象出一个接口来吧

- 人间四月天 2020-07-22 08:42:26
  非常赞同!
  如果把设计原则，设计模式，结合一个开源框架讲解就更到位了，如果自己实现一个具体的框架，在把这些原则和模式结合进去，作为评价这个具体框架的依据，就会理解的非常通透。